Consistency in 3D

Marc Shapiro, Masoud Saeida Ardekani, Gustavo Petri

Consistency in 3D*

Marc Shapiro[1], Masoud Saeida Ardekani[2], and Gustavo Petri[3]

1 **Sorbonne-Universités-UPMC-LIP6 & Inria Paris**

2 **Purdue University** †

3 **IRIF, Université Paris Diderot**

Abstract

Comparisons of different consistency models often try to place them in a linear strong-to-weak order. However this view is clearly inadequate, since it is well known, for instance, that Snapshot Isolation and Serialisability are incomparable. In the interest of a better understanding, we propose a new classification, along three dimensions, related to: a total order of writes, a causal order of reads, and transactional composition of multiple operations. A model may be stronger than another on one dimension and weaker on another. We believe that this new classification scheme is both scientifically sound and has good explicative value. The current paper presents the three-dimensional design space intuitively.

1998 ACM Subject Classification C.2.4 Distributed databases; D.1.3 Concurrent programming; D.2.4 Software/Program Verification; E.1 Distributed data structures

Keywords and phrases Consistency models; Replicated data; Structural invariants; Correctness of distributed systems;

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2016.<article-no>

1 Introduction

A distributed database maintains data scattered and replicated across nodes separated by networks that are inherently slow and unreliable. In this context, designers face an inherent trade-off between system cost and application cost. In particular, the CAP theorem [13] shows that, when failures can partition the network (P), a database can either be strongly consistent (C) or available (A), but not both. Strong consistency masks parallelism and failures from the application, at the cost of constant synchronisation, which translates to high latency and even stalling when the network is down (CP). A model with weaker consistency significantly improves availability, performance and cost (AP), but increases the opportunities for subtle yet potentially catastrophic application errors.

This trade-off has spurred a lot of creativity. A dizzying number of consistency designs are available, as theoretical models, protocol designs, and implemented systems. Note however that, among the many options, not all are related to CAP.

In order to develop high-performance yet correct distributed applications, we need a better understanding, in particular how an application's needs relate to consistency. How

* This research is supported in part by European FP7 project 609 551 SyncFree.

† Now at Samsung Research America

does a particular application behave in a particular consistency model? What are its pros and cons? This paper aims to clarify this crowded space.

The strongest consistency model, called Strict Serialisability (SSER), has three remarkable features:[1] absence of concurrent operations, i.e., transactions execute in a total order;this ordering is monotonic and respects causality; and the unit of interaction with the database, the transaction, is a composition of operations. The thesis of this paper is that each of these features aims to guarantee a different class of application invariants. The mechanism associated with each feature has an inherent associated cost, respectively synchronisation, transitivity, and grouping.

Other consistency models differ from SSER by providing the same three features to a lesser degree, or even not at all. Relaxing a feature generally lowers its system cost but weakens the class of guaranteed invariants, increasing its cost to application programmers. Accordingly, we argue for classifying models in a three-dimensional space, along the axes of total-order, visibility, and composition. This insight is illustrated in Figure 1, and is fully detailed in Table 5.

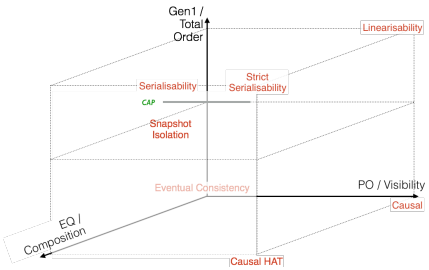

Figure 1 Some consistency models situated in the three dimensions

Whereas previous surveys [1, 2, 29] are comprehensive and detailed, our focus is more pedagogical. Our three axes constitute a simplification. We do not claim to explain everything, but wish to help the reader situate a model on a mental map, glossing over details when convenient.

This paper is structured as follows. After this introduction (Section 1), Section 2 presents a generic system model. Then we define and discuss the three axes in turn: Gen1 invariants and total order in Section 3, PO invariants and visibility order in Section 4, EQ invariants and composition in Section 5. Finally, Section 6 summarises the relations between the three axes and concludes.

Inevitably, it is difficult to discuss one axis without referring to the others. We ask the reader's patience with such apparent circularities, which we do our best to minimise.

2 System model

Our model and definitions are derived from previous work [9, 14, 25, 29]. The system is composed of an unbounded set of sequential processes, uniquely identified. We divide it into an application layer running above a consistency layer. The application consists of *objects* stored in the database and of *client* processes that call operations on objects and receive results in return. Clients do not communicate directly, only via operations on shared data.

The consistency layer manages state and executes the message sending, receiving and delivery events described hereafter. A consistency model consists of a set of restrictions imposed on the ordering of events, in order to *guarantee* a class of *invariants* that remains true in any execution of that model. Ideally, we would like to ensure any invariant of a sequential execution.

[1] We refer to Table 6 for a full list the consistency models discussed herein and the primary reference for each.

(a) Operation u decomposed into indivisible call, return, generator $u_?$ and effector $u_!$. Precondition u_{PRE} is true at the effector.

(b) The effectors of concurrent operations may execute in different orders in the general case.

Figure 2 Operation model

To simplify the discussion, there will be no failures: a message sent is eventually received, unaltered, by its destination process. We focus on safety and do not consider liveness properties.

2.1 Data

Server processes collectively implement the abstraction of a database or persistent memory. The database consists of discrete data items of *objects* x, y. The state of server i, noted σ_i, contains a copy or *replica*, noted x_i, of object x.[2]

A common object type is the *register*, which supports the read and write operations, respectively returning and completely overwriting overwrites the register's content. The state of a register depends only on the last write. However, our model is not restricted to registers. The application may store object types of arbitrary complexity, for instance a set, a stack, a table, or a tree, with their high-level operations (respectively, add and remove, push and pop, insert row, or rebalance).

2.2 Operations

We decompose an operation into indivisible asynchronous events, as illustrated in Figure 2a (a specific consistency model may place restrictions on their ordering).

The semantics of update operations is defined by a function: $\mathcal{F} \in \mathsf{Op} \to (\mathsf{State} \to \mathsf{Val} \times (\mathsf{State} \to \mathsf{State}))$ where Op is the set of operations, State the set of replica states, and Val the set of return values. Operationally, an update u starts as a *call* event, a message from client to origin replica. Delivering this message triggers an initial computation at the origin, called the *generator* $u_?$. The generator reads the state of the origin without modifying it, then: *(1)* Computes a return value u_{RET}, sent back to the client. When the client receives it, the update is *visible to the client*. *(2)* Computes a state transformation, the *effector* $u_!$. The effector is sent to all replicas, including the origin itself. If and when a replica *delivers* the effector, it applies its transformation to the replica's local state, making the update *visible to the replica*.

The effector is generated based on the origin state read by the generator; we abstract this dependence with the precondition u_{PRE} of the effector. The generator can check the precondition at the origin but not at other replicas [14].

[2] To simplify the model, we assume *full replication*: every replica has a copy of every object.

An operation reads (generator) and writes (effector) the replicas of a single object. As we shall see shortly, objects can be connected by invariants and/or transactions.

The history of a client consists of a sequence of call (sending) followed by return (delivery) events. The history of a server consists of a sequence of generator (reading and sending) events and effector (delivery and side-effect) events. The current state of a server can be identified with the sequence the effectors it has delivered. In the general case, effectors of different updates may be delivered in different orders, as illustrated in Figure 2b.

This model is very general. It abstracts away from any specific data type (from registers to complex data types with high level operations), transmission mode (state-based or operation-based), and concurrency semantics (which will be encoded into the function definition). We model operations that do not return by returning nil; we model queries that do not modify state by the skip effector; we abstract arguments away by folding them into the function definition.

The client may choose an arbitrary origin replica, not necessarily the same for successive client operations. Therefore, client-side guarantees may be weaker than those at the server [8, 27]. Conversely, we consider that the server-side guarantees are at least as strong as the client-side ones.

2.3 Executions

We define an execution as a tuple $ex = \langle R, E, \xrightarrow{\mathsf{so}}, \xrightarrow{\mathsf{ro}}, \xrightarrow{\mathsf{ext}} \rangle$ where: *(i)* R is a set of replicas – which shall otherwise remain abstract. *(ii)* E is a set of events, including calls, generators, effectors and returns. *(iii)* $\xrightarrow{\mathsf{so}}$ is a relation among events of the *same session* [29], indicating the order in which the client issued the operations; $\xrightarrow{\mathsf{so}}$ is our abstraction of the behavior of the client. *(iv)* $\xrightarrow{\mathsf{ro}}$ is a family of orders – indexed by replica name – of events that affect that replica. These events include: call, generator and return events for any operation that has this replica as its origin, and effectors of any operation that is delivered to the replica. We shall denote $\xrightarrow[r_i]{\mathsf{ro}}$ the replica order of replica r_i, and we shall overload the notation $\xrightarrow{\mathsf{ro}}$ to denote as the set-theoretic union of the replica orders of each replica, formally the relation $\bigcup_{r_i \in \mathsf{dom}(\xrightarrow{\mathsf{ro}})} \xrightarrow[r_i]{\mathsf{ro}}$. *(v)* Finally, $\xrightarrow{\mathsf{ext}}$ is an *external* order representing the real-time (otherwise called wall-clock time) in which the events occurred; we shall simply assume the existence of this order for some models, and it shall otherwise remain opaque.

Provided with the definition of executions we obtain the derived definition of visibility of operations:

$$u \xrightarrow[r_i]{\mathsf{vis}} v \iff u_! \xrightarrow[r_i]{\mathsf{ro}} v_? \qquad\qquad \xrightarrow{\mathsf{vis}} = \bigcup_{r_i \in R} \xrightarrow[r_i]{\mathsf{vis}}$$

In turn, we obtain a definition the *happens-before* order: $\xrightarrow{hb} = (\xrightarrow{\mathsf{so}} \cup \xrightarrow{\mathsf{vis}})^+$, where we denote by a superscript $^+$ the set-theoretic transitive closure of a binary relation. We speak of *transitive* visibility if $\xrightarrow{\mathsf{vis}}{}^* \subseteq \xrightarrow{\mathsf{vis}}$, and we speak of *causal* visibility if the visibility relation is consistent w.r.t. the happens-before relation: $u \xrightarrow{hb} v \Rightarrow u \xrightarrow{\mathsf{vis}} v$.

2.4 How models relate to application semantics

Applications care about consistency models for the *guarantees* that they provide. We say that model guarantees a certain class of invariants, if an invariant of that class remains

Baseline	Semantic condition	$\Longrightarrow$	Reference
Sequential	Sufficient precondition	Safe	[14]
TOE	Deterministic operations	Same state	[9], [28]
0	Unspecified convergence conditions	EC	[30]
0	Monotonic semi-lattice	Monotonic SEC	[5]
CC	Commutative concurrent effectors	SEC	[25]
CC	Stable effector precondition	Gen1	[14]
SI	Materialized conflict	SI ∩ SER	[11]

0 = lowest point on all three axes. Sequential = sequential non-replicated system.

Table 1 Application assumptions (top) and robustness conditions (bottom).

true in any execution of that model without requiring additional instrumentation from the programmer.

Some guarantees refer to relations between replicas, for instance, Identical State and Convergence. However, an application-observable invariant refers to the state that is observable for a client, e.g., single-object statements such as $x > 0$ or multi-object statements such as $x = y$ or $P(x) \iff Q(y)$. As we shall discuss in detail hereafter, some consistency models guarantee some related classes of invariants.

Although it is convenient to think of the consistency and application levels as independent, this is not entirely true. The correctness of the guarantees rests on two crucial assumptions about the application:

1. A generator and an effector are functions, i.e., their result is deterministic.
2. The application is *sequentially correct*, i.e., each operation (or, in the transactional case, each transaction) in isolation maintains the application invariant. This is the C condition of ACID, called "consistency" or "correctness" in the database literature. Formally, for some invariant I, $\forall \sigma \in \mathsf{State}, u \in \mathsf{Op} : I(\sigma) \wedge u_{\text{PRE}} \implies I(\sigma \bullet u_!)$ for the single-operation case, where we denote by $\sigma \bullet u_!$ the state resulting from updating σ with the effector $u_!$.

A *robustness condition* is one by which an application, running above a less-than-perfect consistency model, can compensate for its deficiencies and support the same invariants as a stronger model. For instance, EC (Eventual Consistency) requires that the application converges, even when running above level zero on all axes (for this, see Convergent Data Types [5] or CRDTs [25]).

Fekete et al. show how the application can emulate Serialisability (SER) above Snapshot Isolation (SI) by applying some simple programming rules [11, Section 5.1].

Gotsman et al. [14] discuss under which conditions concurrent execution can maintain arbitrary application invariants (class Gen1 hereafter). They demonstrate (under certain conditions) that, if all effector preconditions u_{PRE} are stable under all concurrent updates $v_!$, then the invariant remains true, no matter what the order of delivery of effectors.

3 Gen1/Total Order Axis

This section focuses on guaranteeing invariants by restricting concurrency as summarised in Table 2. This axis orders the different consistency properties according to which events must be totally-ordered with respect to each other. Here we consider operations on a single object (the other two axes consider multiple objects). Let us now consider the different

Level	Guarantees	Other axes	Examples
TOG=TOE	Gen1	External visibility Transitive visibility	SSER, LIN SER
Gapless TOE	No lost updates, Identical State	External visibility Causal visibility Transitive visibility	SSI PSI NMSI
Capricious TOE	register $\implies$ Identical State	Transitive visibility	LWW
		Non-monotonic	Bayou
0 = Concurrent	Blind1	Monotonic visibility	RC EC

Table 2 Total-order axis. The double line marks the "CAP boundary."

protocols according to the ordering of events that they impose on the operations to the object of interest (ignoring operations on other objects, which may proceed in parallel).

Unless indicated otherwise, we assume the Monotonic Client property, which is the conjunction of the Monotonic Reads guarantee: given two operations related by the session order $v \xrightarrow{\mathsf{so}} w$, if the former "views" a third operation u (i.e., $u \xrightarrow{\mathsf{vis}} v$) then so does the latter ($u \xrightarrow{\mathsf{vis}} w$); and the Read-My-Writes guarantee: if two operations are related by the session order $u \xrightarrow{\mathsf{so}} v$, then so are they by the visibility order $u \xrightarrow{\mathsf{vis}} v$.

3.1 Same total order for generators and effectors (TOG=TOE)

The first class of models we consider are those for which there exists a Total Order relating all Effectors and Generators (TOG=TOE), let us denote this (existentially quantified) order with the arrow $\xrightarrow{\mathsf{toeg}}$.[3] These are the strongest models in the total-order axis. Evidently, there are a number of constraints that are required for $\xrightarrow{\mathsf{toeg}}$: **1.** Generators and effectors are uninterrupted by other events in the order (therefore the sequences $u_? \xrightarrow{\mathsf{toeg}} v_? \xrightarrow{\mathsf{toeg}} u_!$ and $u_? \xrightarrow{\mathsf{toeg}} v_! \xrightarrow{\mathsf{toeg}} u_!$ are disallowed). **2.** The visibility relation is consistent w.r.t. the total order, meaning that each generator sees exactly the effectors that precede it in this order ($u \xrightarrow{\mathsf{toeg}} v \implies u \xrightarrow{\mathsf{vis}} v$). Table 2 presents in the cell at the first row and last column protocols that fall under this category in the total order axis.

Importantly, the existence of such an order implies that the Visibility relation, restricted to the object, is transitive since each generator must see all the effectors before it, and the effectors of an operation necessarily follow its generator. On the other hand, causality is not guaranteed unless we add the condition that the total order respects the client order ($(\xrightarrow{\mathsf{so}} \cup \xrightarrow{\mathsf{toeg}})^+$ is irreflexive). By adding this additional constraint we require the Visibility relation to be causal for this object.

3.2 TOG=TOE and Gen1 Invariants.

At this strongest point in this axis, we consider generic (arbitrary) single-item invariants, noted hereafter Gen1. For instance, a banking application may require that the balance of

[3] In the interest of readability and space, we shall present some definitions intuitively instead of providing precise mathematical definitions. Their mathematical interpretation is generally self-evident.

accounts be non-negative: $\mathsf{bal} \geq 0$. Another example: an object G that represents a graph, with the invariant that the graph forms a tree.

Recall from Section 2.4 that a sequential program enforces its invariants assuming the effector-precondition u_{PRE}, which may be verified locally by the generator. In the bank account example, $\mathsf{credit(amt)}_!$ and $\mathsf{debit(amt)}_!$ respectively add or subtract amt to or from the local balance bal_i. To maintain invariant $\mathsf{bal} \geq 0$, the sequential preconditions are $\mathsf{credit}_{\text{PRE}} = \mathsf{amt} \geq 0$ and $\mathsf{debit}_{\text{PRE}} = \mathsf{bal} \geq \mathsf{amt} \geq 0$ respectively. However, under unbounded concurrency there exists no safe precondition that can be evaluated locally at the origin replicas; intuitively, enforcing this invariant requires to totally order at least some operations.

In the case of protocols respecting TOG=TOE, and assuming the system respects the ordering of operations issued by the client (the session order: $\xrightarrow{\mathsf{so}}$), any invariant that is correct for a shared memory implementation of the object – where we interpret the clients as being processes, and the database as being the shared memory – will also be respected in this case. We posit that anything provable using, for instance, the Owiki-Gries [21] logic under the shared-memory interpretation, is respected in such a model.

3.3 Total Order of Effectors (TOE): Capricious vs. Gapless

Since generators only read state without changing it, it is tempting to remove them from the total order, therefore allowing concurrency between reads and writes. We shall denote this weaker existential order as $\xrightarrow{\mathsf{toe}}$. This introduces the possibility of anomalies such as write-skew [7].

The order $\xrightarrow{\mathsf{toe}}$ may be *Capricious*: meaning that servers assign sequence numbers independently from one another. While effectors are totally ordered (i.e., each effector has a unique place in the order), they may be received in a non-increasing sequence. This conflicts with the monotonic-client requirement; as a consequence, updates might be lost, if an effector has been delivered at a replica while another effector ordered lower in $\xrightarrow{\mathsf{toe}}$ is received at a later point. This approach is used, for instance, in the Last-Writer Wins (LWW) protocol.

Alternatively, $\xrightarrow{\mathsf{toe}}$ can be *Gapless*: in this case replicas must synchronise to guarantee that the effectors are given a slot in the total order in a strictly monotonic fashion, and therefore replicas can buffer effectors until all prior updates in $\xrightarrow{\mathsf{toe}}$ have been delivered. Here lies the "CAP Line:" Capricious TOE is Available even when the network is Partitioned, whereas Gapless TOE (and of course gapless TOG=TOE) is not Available when Partitions occur.

Strictly speaking, a protocol could be both Capricious and TOG=TOE; however this combination is not very useful; therefore, to simplify the presentation, we order TOG=TOE above Gapless TOE.

3.4 TOE and Causality Based Invariants

In terms of application guarantees provided by protocols satisfying TOE guarantees we cannot generally assume that Gen1 invariants will be satisfied. On the other hand, under Causal Visibility, Rely-Guarantee based techniques can be used [14].

As it is the case with TOG=TOE models, the existence of a $\xrightarrow{\mathsf{toe}}$ order implies that the visibility relation is transitive per-object. If we additionally require that $\xrightarrow{\mathsf{toe}}$ respects

the client order ($\xrightarrow{\text{so}}$) we can conclude that visibility is causal per-object. An important distinction between capricious and gapless $\xrightarrow{\text{toe}}$ models is that in the latter, any two replicas that have received the same updates have the exact *same state*. In contrast, capricious models cannot guarantee the same-state property.

3.5 Concurrent Effectors

At the weakest end of the total order axis, the protocol "Concurrent effectors" in Table 2 does not require any total ordering of effectors and/or generators.

Consider a register, with an invariant that refers only to the current state: e.g., register z must contain an odd number of "1" bits. To maintain it, the order and history of updates is immaterial, and it suffices that each update is individually safe. We shall denote these invariants that are *blind* to the environment and on a single object (1), Blind1 in Table 2.

4 PO/Visibility Axis

The Visibility dimension (Table 3) constitutes our second axis. It aims to guarantee invariants that require control of which effectors are visible, in which order, to generators. Whereas the first axis concerned single-object guarantees, this one connects multiple updates, system-wide. Whereas the first axis is concerned mostly about writes (effectors), this one is mostly about reads (generators). However, they are not totally independent.

4.1 PO-type invariants

The PO-type invariants discussed in this section abstract the concept of a partial order. Conventionally we will write them as $L \geq R$ and refer to the two terms as left- and right-hand-side, LHS and RHS, respectively. The prime example is program order, where each process proceeds through statements $S_1; S_2; \ldots; S_n$, left to right. This may be re-written (abusing notation somewhat) as $S_1 \Longleftarrow S_2 \Longleftarrow \ldots \Longleftarrow S_n$, i.e., executing S_i implies that S_{i-1} has executed. Similarly, write-read dependences, where $v_?$ reads the result of $u_!$, can be summarised as $u_! \Longleftarrow v_?$ and message delivery as $u_? \Longleftarrow v_!$.

Other PO-type invariants are traditional data invariants, such as "employee's salary must be less than his manager's", stock maintenance [6], or referential integrity (object x allocated $\Longleftarrow$ y points to x).

Even with unbounded concurrency, it is safe to update the objects involved in a PO-type invariant, by *first* increasing the LHS by some amount c, and *later* increasing the RHS by an amount $c' \leq c$. More generally, it is always safe to strengthen the invariant, and later weaken it assuming that the prior strengthening has been applied. This is known as the Demarcation Protocol [6] or the safe-publication idiom [1]. As a special case, c' can be null, i.e., it is safe to unilaterally increase the LHS.

We will consider different versions of the demarcation protocol according to the visibility guarantees enforced by the underlying model (as shown in Table 3). For instance, for a system enforcing causal visibility, we can operate under the causal demarcation protocol if one client does the strengthening of the invariant and notifies another client of this fact by writing on a flag. When the second client sees the effects of the update on the flag, by causality we can assume that the invariant can be safely weakened according to the

Level	Guarantees	Other axes	Example
External	external demarcation		SSER, LIN, SSI
Trans. Vis. + Client Order = Causal Visibility	causal demarcation	TOE not TOE	PSI Causal HAT, CC
Monotonic + WR dependence = Transitive Visibility	transitive demarcation	TOE not TOE	SER, NMSI
MR + RMW = Monotonic Client	client progress		
0 = Rollbacks			Bayou

MR = Monotonic Reads. RMW = Read-My-Writes. WR = Write-Read dependency. Client Order conjoins all these relations with Write-Write dependencies [29].

Table 3 Visibility axis

prior strengthening on the other operation. A similar arguments can be made for transitive demarcation.

The above requires that updates become visible to other replicas in the same order. We discuss such protocols in the next section.

At the weakest level of the Visibility axis, labeled "Rollbacks" in Table 3, there is no required order between reads. A client could observe the effects of some update u, and later observe a state where u has not occurred. This violates the so-called Monotonic-Reads session guarantee [27]. Similarly, a client might update an object, and later observe a state of the object before the update is applied. This violates Read-My-Writes [27].

Few systems are at Rollback level; most models assume what we call the Monotonic Client level, in which the client state is monotonic, ensuring both Monotonic Reads and Read-My-Writes (as defined in Section 3). In fact, client monotonicity must appear so obvious that many authors do not even state this assumption, e.g., Gray and Reuter [15]. We will follow the common practice of assuming the Monotonic Client guarantee in this paper, unless explicitly mentioned. Frigo [12] argues that non-monotonic models are "not reasonable," but some systems deliberately eschew these guarantees for the benefit of responsiveness [28].

The next-stronger level, Transitive Visibility, simply requires the visibility relation to be transitive. Given operations u and v, if the (generator of) update v reads the result of (the effector of) update u, then all clients should observe the results of u before those of v. Formally $\xrightarrow{\mathsf{vis}}{}^{*} \subseteq \xrightarrow{\mathsf{vis}}$. Note that Total Order of Effectors implies Transitive Visibility, but not vice-versa. Not all models have the Transitive Visibility property. For instance, SER has it, but not EC nor PRAM. To simplify the presentation, hereafter Transitive Visibility also includes Monotonic Client.

The next level adds Client Order (Monotonic Writes and Writes Follow Reads [27]), resulting in Causal Visibility (also called Causal Consistency or Causal Memory [3]). Formally this requires that visibility be consistent with the session order: $\xrightarrow{hb} \subseteq \xrightarrow{\mathsf{vis}}$. Transitive and Causal Visibility are partial orders. They can be further strengthened by requiring the existence of a *total* order that is causal (hence also transitive); this point meets the TOG=TOE point of the Total Order axis of Section 3.

Causal visibility is strictly stronger than transitive visibility, and is not supported by all models. As a case in point, SER does not require causal visibility: if a client calls operations u and then v, and u and v are on different object, a server (even the origin server) may

execute v before u.[4]

The highest point in the Visibility axis is External Consistency. This requires that all operations are totally ordered (finding a $\xrightarrow{\mathsf{toeg}}$ as in Section 3), and that this order coincides with the external (real-time) order: $\xrightarrow{\mathsf{ext}} \subseteq \xrightarrow{\mathsf{toeg}}$. In this way, updates can be related with external events, and the causality between internal and external events is preserved.

Causal Visibility is the conjunction of the four so-called session guarantees [8]: formally, all sixteen combinations are possible. Pragmatically, however, we find that the linear presentation of Table 3 captures the important practical properties.

5 EQ/Composition Axis

Our third axis aims to guarantee some form of coupling between separate objects. It provides mechanisms to: *(i)* compose together multiple updates and multiple objects dynamically, and *(ii)* to close the guarantees provided by the Total Order and/or Visibility axes over the whole composition.

5.1 EQ-type and Gen* invariants

An EQ-type invariant is one that maintains an equivalence relation between objects. EQ requires to always group together updates to both objects; we call this All-Or-Nothing Effectors; intuitively, either all the updates of the composition are visible, or none is. For instance, a symmetric friendship graph $x.\mathsf{friendOf}(y) \iff y.\mathsf{friendOf}(x)$, or disjoint union to a constant set, $A \cap B = \emptyset \wedge A \cup B = C$. Notice the similarity between EQ and Blind1: neither depends on previous state, only on the current transaction (resp. operation). As it is the case for Blind1 invariants, in order to verify EQ invariants, no ordering assumptions are required from the environment, and it suffices to show that each individual transaction preserves the invariant if it was initially valid.

Consider now a generic sequential invariant over multiple objects, noted Gen*. Since multiple objects are involved, this likely requires All-Or-Nothing Effectors. Furthermore, the generators' reads will need to be mutually consistent, and served from a *consistent snapshot*. Finally, Gen* may require a total order, by the same reasoning as for Gen1 (recall Sections 3.2 and 3.3). The Transactional Composition axis serves to enforce these requirements.

Transactions support "ad-hoc" composition. For instance, when buying a ticket online, ensuring that the buyer has sufficient balance and that a ticket is available (ad-hoc Gen*), and ensuring that the money is both debited from the buyer's account and credited to the seller's (ad-hoc EQ).

[4] Here we argue about operations, while serialisability is defined for transactions. The analogous argument is obvious assuming that the transactions operate on different object sets.

Level	Guarantees	Other axes
All-or-Nothing + Snapshot	EQ + Gen*	TOG=TOE
	EQ	
All-or-Nothing Effectors		
0 = Single Operation		

Table 4 Composition axis

5.2 (Transactional) Composition axis

For this axis we add begin and end markers to the repertoire of events uttered by a client, grouping all the intervening calls and returns into one transaction. Depending on the model, transactions may be associated with the properties "All-Or-Nothing Effectors" and "Snapshot." Table 4 shows the composition axis.

In many implementations, a server may execute a transaction speculatively, and either commit or abort at the end [23]. An aborted transaction has no effect and does not return anything. Our model considers only committed transactions.

All-Or-Nothing Effectors means that, if some effector of transaction T1 is visible to transaction T2, then all of T1's effectors are visible to T2. (This is the A in ACID, sometimes called Atomic.) TOE guarantees extend to all effectors of a transaction: if $u_!$ and $v_!$ are part of T1, $w_!$ and $t_!$ are of part of T2, and $u_! < w_!$ in the TOE, then $v_! < w_!$ and $u_! < t_!$. We may write simply $\mathrm{T1}_! < \mathrm{T2}_!$.

Typically, all the generators of the transaction read from a same set of effectors, called its *snapshot*. Generator order guarantees, if any, extend to the whole snapshot, i.e., *(i)* Monotonic-Client, resp. Transitive, resp. Causal Visibility: the snapshot (the set of effectors read from) is closed under the visibility order. *(ii)* TOG=TOE: the generators are adjacent in the total order.

5.3 Composition: Discussion

Transactional protocols generally assume All-or-Nothing but differ in their snapshot guarantees. For instance, SER, NMSI or SI require Transitive Visibility but do not enforce client order, i.e., Monotonic Writes [15]. Indeed, these models allows a client to execute T1; T2 and the system to serialise as T2; T1 if their read-write sets are disjoint. Strong Snapshot Isolation (SSI) does ensure client order, hence Causal Visibility, as it mandates to choose a snapshot greater than any commit point when a transaction starts. The same is true of a protocol that requires external causality, such as Strict Serialisability (SSER).

In addition to the features discussed so far, snapshots may be partially ordered or totally ordered. For instance, NMSI's snapshots are partially ordered, whereas SI, SSI, and SER snapshots are totally ordered. This represents the main difference between SI and NMSI.

As a simplification, our linear axis does not differentiate between partially- and totally-ordered snapshots. Unfortunately and consequently, SI is missing from our summary table (Table 5) as it would occupy the same position as NMSI.

Total Order	Composition	Visibility				
		Rollbacks	Monotonic	Transitive	Causal	External
TOG=TOE	All-or-Nothing + Snapshot			SER		SSER
	All-or-Nothing Effectors					
	Single Operation				SC	LIN
Gapless TOE	All-or-Nothing + Snapshot			NMSI	PSI	SSI
	All-or-Nothing Effectors					
	Single Operation					
Capricious TOE	All-or-Nothing + Snapshot	Bayou				∅
	All-or-Nothing Effectors					∅
	Single Operation		LWW			∅
Concurrent Ops	All-or-Nothing + Snapshot				Causal HAT	∅
	All-or-Nothing Effectors		RC			∅
	Single Operation	EC	PRAM		CC	∅

Table 5 Matrix of features and consistency models

6 Discussion and conclusion

Our system model (Section 2) is very general. The separation between generators and effectors allows for internal parallelism; if unusual, it reflects practical implementations [23]. Our total order axis (Section 3), classifies the degree of concurrency between operations to a single object, including only effectors or also generators, and accounts for both available (capricious) and consensus-based (gapless) approaches. The other two axes introduce mechanisms that relate multiple objects; however, they serve different purposes and have different costs. Visibility order (Section 4) relates reads to writes and involves maintaining a system-wide transitive closure, and aims to support PO-type invariants. Transactions (composition, Section 5) serves to enforce ad-hoc EQ and Gen*; a transaction is a one-off grouping, requested by the application.

In order to be intuitively useful, our classification simplifies the design space into three approximately linear axes (which we relate to *application invariants*). Obviously, this cannot account for the full complexity of the relations between models. We acknowledge the deficiencies of such a simplification. For instance, we flatten the visibility axis, and abusively assume that all TOG=TOE models must be gapless. We defend this simplification as practically relevant, even if not formally justified. We also ignored hybrid models, such as Update Serialisability [16].

We focus on client-monotonic models, as they are the most intuitive, and because monotonicity is trivial to implement. While the specifications of SER, NMSI, or RC do not require Monotonic visibility, all the actual implementations that we know of do provide it.

Table 5 positions some major consistency models within the three axes. Compare for instance two prominent strong consistency models: SSER and LIN. While LIN considers single operations and single objects, SSER is a transactional model requiring All-or-Nothing and Snapshot. Also notice how the visibility axis differentiates SSER from SER, and NMSI from PSI.

While our results are preliminary, we believe that this classification sheds light on the crowded space of distributed consistency guarantees, towards a better understanding of the application invariants enforced by each of them. We intend, in further work, to formalize our definitions and prove some interesting meta-properties. This work aims to be an step

Acronym	Full name	Type	Total-Order	Visibility	Composition	Ref.
Bayou	Bayou	system	Capricious TOE	Rollbacks	All-or-Nothing + Snapshot	[28]
CC	Causal Consistency	model	Concurrent Ops	Causal	Single Operation	[3]
Causal HAT	Causal Highly-Av. Txn.	model	Concurrent Ops	Causal	All-or-Nothing + Snapshot	[4]
EC	Eventual Consistency	model	Concurrent Ops	Rollbacks	Single Operation	[30]
LIN	Linearisability	model	TOG=TOE	External	Single Operation	[17]
LWW	Last-Writer Wins	protocol	Capricious TOE	Monotonic	Single Operation	[18]
NMSI	Non-Monotonic SI	model	Gapless TOE	Transitive	All-or-Nothing + Snapshot	[24]
PRAM	Pipeline RAM	model	Concurrent Ops	Monotonic	Single Operation	[20]
PSI	Parallel SI	model	Gapless TOE	Causal	All-or-Nothing + Snapshot	[26]
RC	Read Committed	model	Concurrent Ops	Monotonic	All-or-Nothing Effectors	[7]
SC	Sequential Consistency	model	TOG=TOE	Causal	Single Operation	[19]
SER	Serialisability	model	TOG=TOE	Transitive	All-or-Nothing + Snapshot	[15]
SI	Snapshot Isolation	model	Gapless TOE	Transitive	All-or-Nothing + Snapshot	[7]
SSER	Strict Serialisability	model	TOG=TOE	External	All-or-Nothing + Snapshot	[22]
SSI	Strong Snapshot Isolation	model	Gapless TOE	External	All-or-Nothing + Snapshot	[10]

Table 6 Cross-reference of models, protocols and systems

towards a rigorous and systematic understanding of distributed database implementations and their applications.

References

1 Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

2 Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Mass. Institute of Technology, Cambridge, MA, USA, March 1999.

3 Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995.

4 Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, November 2013.

5 Carlos Baquero and Francisco Moura. Using structural characteristics for autonomous operation. *OSR*, 33(4):90–96, 1999.

6 Daniel Barbará-Millá and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *VLDB Jrnl.*, 3(3):325–353, July 1994.

7 Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

8 J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, pages 152–158, A Coruña, Spain, February 2004. Euromicro.

9 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *POPL*, pages 271–284, San Diego, CA, USA, January 2014.

10 Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *VLDB*, pages 715–726, 2006.

11 Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *TODS*, 30(2):492–528, June 2005.

12 Matteo Frigo. *The weakest reasonable memory model*. PhD thesis, MIT, Cambridge, MA, USA, October 1997.

13 Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

14 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *POPL*, pages 371–384, St. Petersburg, FL, USA, 2016.

15 Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco CA, USA, 1993. ISBN 1-55860-190-2.

16 R C Hansdah and Lalit M. Patnaik. Update serializability in locking. In Giorgio Ausiello and Paolo Atzeni, editors, *Int. Conf. on Database Theory*, pages 171–185, 1986.

17 Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

18 Paul R. Johnson and Robert H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, January 1976.

19 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

20 R J Lipton and J S Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, 1988.

21 Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *CACM*, 19(5):279–285, May 1976.

22 Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

23 F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *J. of Dist. and Parallel Databases and Technology*, 14(1):71–98, 2003.

24 Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, pages 163–172, Braga, Portugal, October 2013. IEEE Comp. Society.

25 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *SSS*, volume 6976 of *LNCS*, pages 386–400, Grenoble, France, October 2011. Springer Verlag.

26 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *SOSP*, pages 385–400, Cascais, Portugal, October 2011. Assoc. for Comp. Mach.

27 Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, pages 140–149, Austin, Texas, USA, September 1994.

28 Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–182, Copper Mountain, CO, USA, December 1995. ACM SIGOPS, ACM Press.

29 Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. ArXiv e-print 1512.00168, arXiv.org, December 2015. Accepted for publication in ACM Computing Surveys.

30 Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008.